NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

REMOTE APPLICATION SUPPORT IN A MULTI LEVEL ENVIRONMENT

by

Robert C. Cooper

March 2005

Thesis Co-Advisor:	Thuy D. Nguyen
Thesis Advisor:	Cynthia E. Irvine

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** March 2005	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE: Remote Application Support In A Multilevel Environment	**5. FUNDING NUMBERS**
6. AUTHOR(S) Robert C. Cooper	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

The use of specialized single-level networks in current military operations is inadequate to meet the need to share information envisioned by the Global Information Grid (GIG). Multilevel security (MLS) is a key Information Assurance enabler for the GIG vision. The Monterey Security Architecture (MYSEA), a distributed MLS network, eliminates the need to use separate equipment to connect to many networks at different classification levels. It allows users to view data at different sensitivities simultaneously. MYSEA also allows commercial software and hardware to be used at clients.

To address the threat of residual data on the client after a user session change in security state, the MYSEA clients are required to be "stateless", i.e., there is no non-volatile writable memory. Hence the MYSEA server must provide the clients with the ability to execute server-resident client-side applications to access data at different security levels over the MLS Local Area Network (LAN). The MYSEA server currently does not support such capability. This thesis addresses this limitation. A new trusted process family is introduced to provide a pseudo-socket interface for the single level remote application to access the MLS LAN interface. Detailed design specifications were created to facilitate implementation of the remote application support.

14. SUBJECT TERMS Multilevel Security (MLS), Information Assurance (IA), Monterey Security Architecture (MYSEA), Client-Side Remote Application, Trusted Remote Session Management	**15. NUMBER OF PAGES** 77
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

THIS PAGE INTENTIONALLY LEFT BLANK

REMOTE APPLICATION SUPPORT IN A MULTI LEVEL ENVIRONMENT

Robert C. Cooper
Lieutenant, United States Navy
B.S., Rensselaer Polytechnic Institute, 1996

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2005

Author:	Robert C. Cooper
Approved by:	Cynthia E. Irvine, Ph.D. Thesis Advisor
	Thuy D. Nguyen Co-Advisor
	Peter J. Denning, Ph.D. Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The use of specialized single-level networks in current military operations is inadequate to meet the need to share information envisioned by the Global Information Grid (GIG). Multilevel security (MLS) is a key Information Assurance enabler for the GIG vision. The Monterey Security Architecture (MYSEA), a distributed MLS network, eliminates the need to use separate equipment to connect to many networks at different classification levels. It allows users to view data at different sensitivities simultaneously. MYSEA also allows commercial software and hardware to be used at clients.

To address the threat of residual data on the client after a user session change in security state, the MYSEA clients are required to be “stateless”, i.e., there is no non-volatile writable memory. Hence the MYSEA server must provide the clients with the ability to execute server-resident client-side applications to access data at different security levels over the MLS Local Area Network (LAN). The MYSEA server currently does not support such capability. This thesis addresses this limitation. A new trusted process family is introduced to provide a pseudo-socket interface for the single level remote application to access the MLS LAN interface. Detailed design specifications were created to facilitate implementation of the remote application support.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ABBREVIATIONS AND ACRONYMS

APS	Application Protocol Server
DAC	Discretionary Access Controls
MAC	Mandatory Access Controls
MLS	Multilevel Security
MYSEA	Monterey Security Architecture
PSKT	Pseudo-Socket
RA	Remote Application
RAPSKT	Remote Application Pseudo-Socket
SARP	Secure Attention Request Packet
SSD	Secure Session Daemon
SSS	Secure Session Server
STOP	Secure Trusted Operating Program
TCB	Trusted Computing Base
TPE	Trusted Path Extension
TPS	Trusted Path Server
TRSS	Trusted Remote Session Server

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank my thesis advisors, Dr. Cynthia Irvine and Thuy Nguyen, for all their support on this project. I would also like to thank Thuy Nguyen, David Shifflett and Jean Khosalim for their guidance and insight with the design specification for the remote application support and assistance with the XTS-400 system.

This work was sponsored in part by the Office of Naval Research and the National Reconnaissance Office. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of either the Office of Naval Research or the National Reconnaissance Office.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. MOTIVATION FOR STUDY

Currently, multiple networks running at different security levels must be used to protect information with different classifications. To view data at different classification levels, it is necessary to have separate equipment to connect to all different networks, or to move data from the lower classification levels to the highest level network. The multilevel security (MLS) paradigm eliminates the need for maintaining specialized networks.

The vision of the Global Information Grid (GIG) is to provide a globally interconnected network-centric information environment in order to afford information superiority to the warfighter [1]. This network will require a high-level of information assurance to protect information with different classifications. This information needs to continue to be protected as it is sent remotely to the warfighter in the battlefield. The use of MLS in the GIG is also more efficient than using specialized networks running at different security levels.

The Monterey Security Architecture (MYSEA) MLS network is currently being developed to provide a high assurance system that enforces multilevel security policies. The design includes a limited but sufficient number of high-assurance elements and commercial low-assurance elements. The protection mechanisms ensure that users can only access information that they are currently authorized to access. It also allows commercial software and hardware to be used at clients, thus making available to users the unmodified computing functionalities to which they are accustomed. [2]

However, not all the requirements necessary for the MYSEA MLS network have currently been met. There is a need for MYSEA clients to have the capability to run remote client-side applications on the MYSEA MLS server over the MYSEA MLS Local Area Network (LAN). This remote application support is necessary to allow “stateless” clients to use server-resident applications to access data at different security levels.

B. OBJECTIVE

The objective of this research was to complete a detailed design for the remote application support. Detailed design specifications, based on the pre-existing MLS LAN design developed in previous work, were produced. In addition, a sketch of a testing methodology using the Trivial File Transfer Protocol (TFTP) client program was investigated.

A number of criteria were considered when designing the remote application support. The remote application services had to be decomposed into well-defined modules. The design would undergo extensive reviews by the MYSEA engineering team. The amount of trusted code necessary for the support was to be only what is necessary and sufficient for the remote application support to function. The remote application support would allow the remote application to be used with a limited number of changes.

C. ORGANIZATION OF THESIS

This thesis is organized as follows: Chapter I gives a brief introduction to the project including a brief MYSEA introduction, the high level objective of this work, and the motivation behind this research. Chapter II provides the background for this work. This includes a description of the MYSEA architecture, an overview of the existing MYSEA design and information on XTS-400 restrictions and features that affect the design. Chapter III describes the overall concept of how the remote application support will operate. It also defines the functional requirements that the design must satisfy. Chapter IV describes how the remote application support was designed and how the new components interact with the pre-existing ones. Chapter V provides some possible enhancements which could be added to the design and some suggestions for implementation and testing. It then concludes with a few final thoughts on the project. Appendix A lists the contents of the updated design specifications.

II. BACKGROUND

A. INTRODUCTION

This chapter contains background information on the Monterey Security Architecture (MYSEA), the pre-existing modules in the Multilevel Security (MLS) Local Area Network (LAN) design and the XTS-400.

B. MONTEREY SECURITY ARCHITECTURE (MYSEA)

The Monterey Security Architecture (MYSEA) provides a distributed network architecture which includes a limited but sufficient number of high-assurance elements for enforcing multilevel security policies. The majority of the components come from low-assurance commercial elements. The architecture consists of three main components: MYSEA Servers, MYSEA Clients and pre-existing single level networks. The MYSEA server and MYSEA clients co-exist on an MLS Local Area Network (LAN) [2].

1. MYSEA Server

The MYSEA server runs on a DigitalNet XTS-400 Trusted Computer System and includes both a Trusted Computing Base (TCB) and untrusted policy-aware Application Protocol Servers (APS). The DigitalNet Secure Trusted Operating System (STOP), which enforces multilevel security policies, forms the basis of the TCB. The functionality of the XTS-400 TCB has been extended to include multilevel trusted path services and multilevel secure session services. The trusted path services are used to support trusted remote authentication by providing a trusted mechanism for communication between the client and the server. The secure session services are used to start untrusted APSs and client-side remote applications. The secure session services assign the APSs and RAs the security level that has been negotiated through the trusted path service mechanism.

The trusted path services and secure session services are currently provided by the Trusted Path Server (TPS) and Secure Session Server (SSS) respectively. More detail on both of these servers and the APS can be found in Section D below.

2. MYSEA Client

MYSEA clients are untrusted commercial-off-the-shelf personnel computers. Since the intent is for users to be able to log into the server at different security levels, the clients must be stateless in order ensure that object reuse requirements are met, i.e. that no information related to the previous security level remains on the untrusted PC. The operating system and applications are loaded into volatile RAM from a non-writeable source with user preferences stored on the MYSEA server. The client connects to the MYSEA server using a Trusted Path Extension (TPE) located at each client. The TPE provides a secure unforgettable connection between the MYSEA client and server which does not depend upon the security of the MYSEA clients.

The network interface on the server connected to the MYSEA clients is configured to the level of the MLS LAN. This means that any process running on the server that requires access to this network interface will also have to run at the level of the MLS LAN. The pre-existing processes that need to run at this level include the TPS Parent Process, the TPS Child Process, the SSS Parent Process and the SSS Child Process. The TRSS Parent and Child Processes described in Chapter IV will also need to run at this level.

3. Single Level Networks

Pre-existing single level networks operating at different security levels are connected to the MYSEA server either through a separate single level network interface or via a multilevel interface. In the first case, a Trusted Channel Module (TCM) is used to provide a secure connection between the MYSEA server and multiple single level networks. In addition, high assurance encryption devices can also be used, if required. In the future, users on single level networks will have access to content on the MYSEA server and MYSEA clients will have access to content on the single level networks without the use of middleware web portal services.

C. XTS-400 SPECIFICS

Information on the XTS-400 can be found in a number of references including a trusted facility manual [3], a programmer’s guide [4] and a user’s manual [5]. This section discusses some of the XTS-400 specifics which affect the design specification.

1. TCP/IP Privileged Ports

The XTS-400 restricts normal user programs from using TCP/IP privileged ports. Only processes whose process identifier equals the network user identifier can use these privileged ports. Only trusted processes may be given the privilege to change the user identifier on the XTS-400 system [3].

This restriction greatly influenced the design of the MYSEA server software. Trusted processes were introduced to service TCP/IP privileged ports on behalf of the APS and RA Processes. This is because the APSs and RAs execute at the session level of the logged in TPE and do not have the privilege to change the user identifier, they cannot use privileged ports directly. Allowing the APSs and RAs to change user identifiers would require them all to be trusted processes which would increase the amount of trusted code necessary on the system.

2. Trusted and Untrusted Processes

Trusted processes on the XTS-400 are able to manipulate Trusted Computing Base (TCB) databases or have privileges which exempt the process from access control rules [3].

The MYSEA server design utilizes these XTS-400 features in order to implement the MYSEA-specific confidentiality and integrity security policies. The SSD, the SSS and the TPS are all trusted processes since they require specific privileges which are listed in the process Section D below. The Trusted Remote Session Server (TRSS) will also need to be a trusted process as described in Chapter IV.

3. System Calls

Many of the standard Linux calls are available on the XTS-400 and can be used by programs running that are hierarchically dependent on the STOP. There are also XTS-400 specific system calls and some of these are only available to trusted processes. The Privileges Module, described in Section D, uses some of the XTS-400 specific calls that can only be used by trusted processes to change privileges. In addition, some system calls have additional functionality when called by a trusted process. For example, a system call which reads an object may provide the ability to read objects at different security levels when invoked by trusted processes [4].

D. PRE-EXISTING DESIGN SPECIFICATION/IMPLEMENTATION

Many of the modules in the pre-existing design originated from the work done by BryerJoyner and Heller [6]. Their work contains the requirements, design and implementation for the original User Identification and Authentication, Buffer IO, Semaphore, Privileges, Shared Memory, Utility, Pseudo-Socket (PSKT), Trusted Path Server (TPS) and Secure Session Server (SSS) modules. These modules have all been changed, expanded and redesigned multiple times. The changes were reflected in an updated design specification draft [7]. The draft also added the Allowed Protocols, Allowed Trusted Path Extension (TPE), PSKT Map, User and Secure Session Daemon (SSD) modules. The most recent version of the design specification includes the new design for client-side remote application support and additional details on the pre-existing modules. The remainder of this section reviews the high-level details of the various pre-existing modules.

1. Overview

The diagram below depicts the sequence of events that take place when a user wants to access an application server running on the MYSEA server.

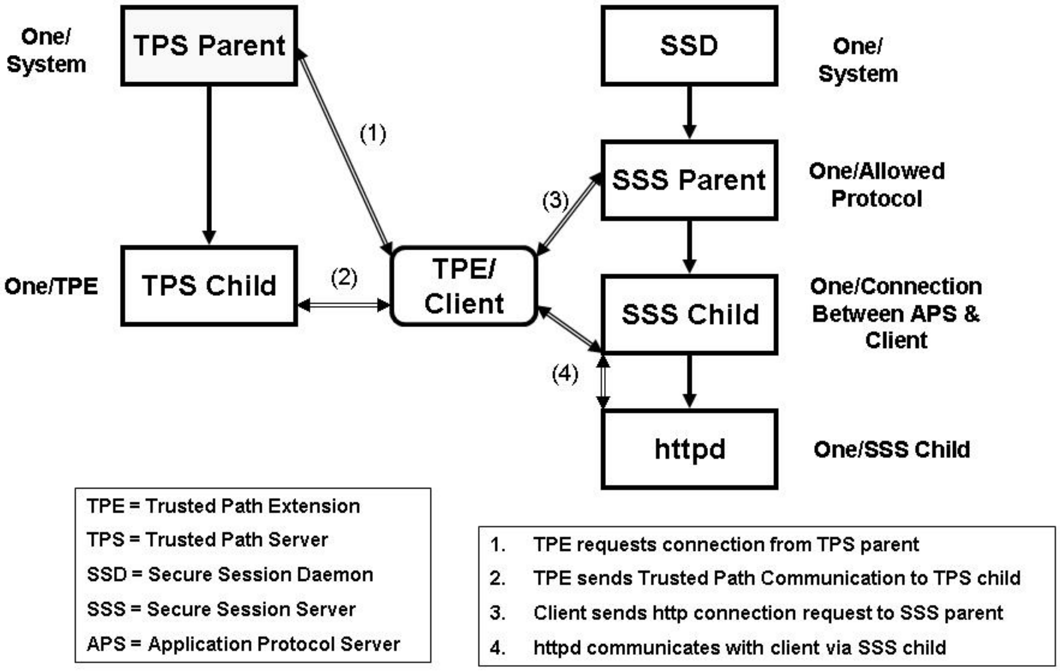

One/ System
TPS Parent
(1)
SSD
One/ System
SSS Parent
One/Allowed Protocol
(3)
One/TPE
TPS Child
(2)
TPE/ Client
SSS Child
One/Connection Between APS & Client
(4)
httpd
One/SSS Child
TPE = Trusted Path Extension
TPS = Trusted Path Server
SSD = Secure Session Daemon
SSS = Secure Session Server
APS = Application Protocol Server
1. TPE requests connection from TPS parent
2. TPE sends Trusted Path Communication to TPS child
3. Client sends http connection request to SSS parent
4. httpd communicates with client via SSS child

Figure 1. Pre-Existing Design Overview

The TPS family of processes, including the TPS Parent and TPS Child, is trusted. The TPS Child Process requires the privilege to access system identification and authentication information.

The SSS family of processes, including the SSD, the SSS Parent and SSS Child, is also trusted. The privilege set for this family of processes includes the ability to change the user identifier of the process and MAC/DAC exemption. Both privileges are invoked by the SSD Process to cleanup up any left over PSKT Databases during startup that could be at any user/session level. The SSS Child Process invokes both privileges to create the PSKT Database used for communications between the SSS Child Process and APS Process. The SSS Child Process also invokes the MAC/DAC exemption privilege to access the PSKT Database since the SSS Child Process normally runs at the level of the MLS LAN, which allows access to the network interface card.

When the MYSEA server starts, the TPS Parent Process and SSD begin execution. After initialization the TPS Parent Process waits for trusted path connection requests. The SSD creates a SSS Parent Process to handle connection requests for each protocol server to be run on the MYSEA server.

When the user desires to access one of the application servers, the user must first invoke the Secure Attention Key on the Trusted Path Extension (TPE) device, which establishes a secure connection with the server. The TPE is then used to login to the MLS server by entering a user name and password in response to prompts issued by the server. A TPS Child Process is created and it uses its privilege to access system identification and authentication information to verify the user name and password. Still using the TPE, the user then chooses an initial session level. The session is finally started by issuing the *run* command on the TPE device. After the session-level connection between the MLS server and TPE is established, the client system is permitted to send data to and receive information from the server via the TPE.

The client can now, for instance, send a request for a web page to the MYSEA server. The SSS Parent Process receives this request and starts an SSS Child Process to receive further http protocol requests from the client. The SSS Child Process uses its privileges to create a PSKT Database to be used for communications between the SSS Child Process and the HTTP server (httpd). The SSS Child Process starts an httpd to handle the HTTP request. The httpd process communicates the results of the request back to the client via the SSS Child Process using the PSKT Database.

2. Databases

The following databases were initially designed to store the data used by the TPS and SSS: Allowed TPE Database, User Database, Allowed Protocols Database, PSKT Map Database and PSKT Database. The diagram below depicts which databases are used by which process during runtime. The type of access (read and/or write) is also included in the diagram. An overview of each database follows.

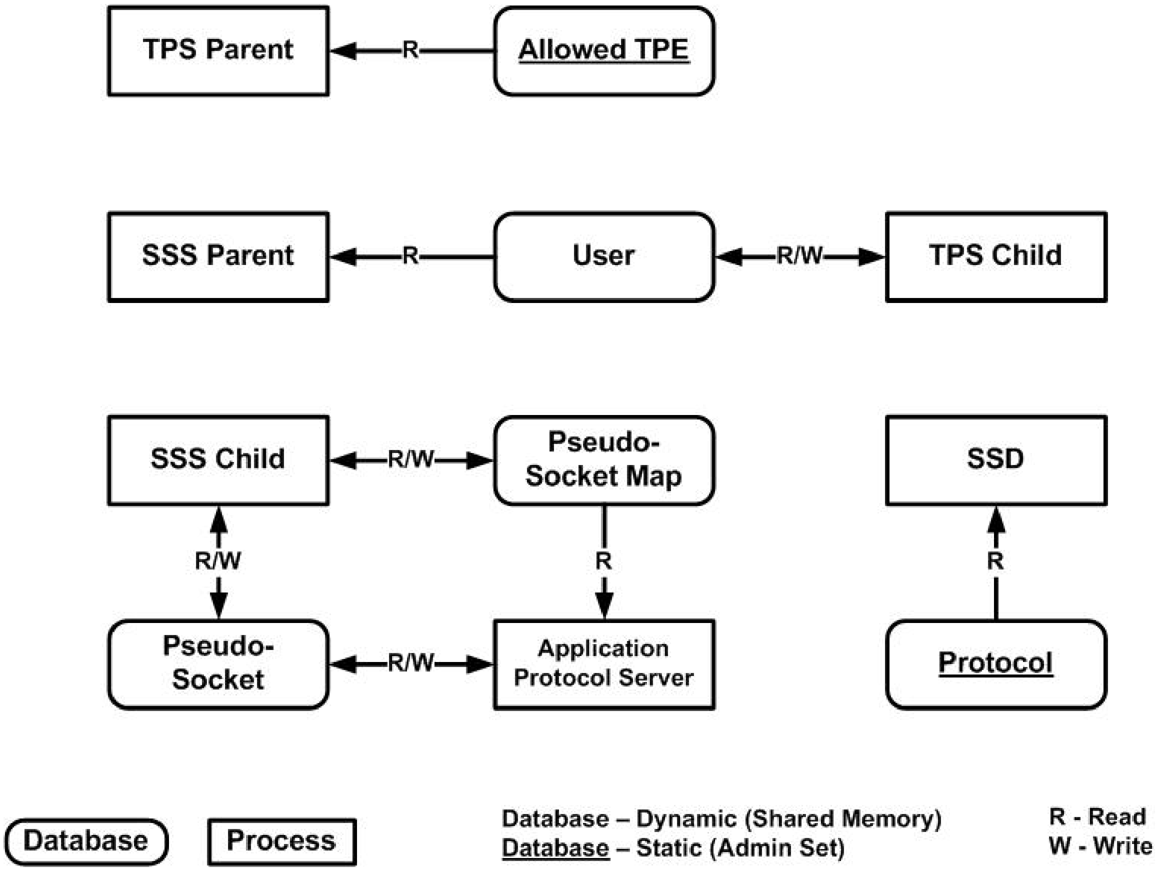


Figure 2. Pre-Existing Process / Database Relationships

a. Allowed Trusted Path Extension (TPE) Database

The Allowed Trusted Path Extension (TPE) Database contains a list of unique identifiers of authorized TPEs. The database is statically configured by an administrator before starting the MYSEA server. The TPS Parent Process uses this database to determine if a TPE is allowed to login to the system.

b. User Database

The User Database is used to keep track of the TPEs that are currently logged into a system. Entries in the database are dynamically updated during runtime. Each entry in the database associates a TPE identifier to the user identifier logged in from that TPE and the session level of the login.

The TPS Child Process has read/write access to the database. When a user logs into the MLS LAN from a TPE, the TPS Child Process adds an entry to the database. The session level is updated if the user changes session levels. The TPS Child Process also removes the entry from the database when the user logs out of the system.

The SSS Parent Process has read access to the database. The SSS Parent Process uses the database to determine the user identifier and session level logged in from a TPE when the process receives a protocol connection request.

c. Allowed Protocols Database

The Allowed Protocols Database contains a list of application protocols that are started by the MLS server during startup. The database is statically configured by an administrator before starting the MYSEA server. Each entry in the database contains an identifier for the protocol, the port number used for listening to requests and a path to the executable for the service. The SSD uses this database during startup to determine which protocols need to be supported. The SSD starts a SSS Parent Process for each protocol in the database.

d. Pseudo-Socket (PSKT) Map Database

The Pseudo-Socket (PSKT) Map Database maps a user ID and session level to a shared memory key which is used to find the shared memory segment that contains a Pseudo-Socket Database. Entries in the database are dynamically updated during runtime. The entries keep track of which shared memory locations are being used for communications between different SSS Child Process and Application Protocol Server (APS) Process pairs.

In the original design, the SSS Child Process had read/write access to the database. When an SSS Child Process was started to handle protocol server requests from a TPE, the SSS Child Process would reserve a PSKT handle in the database by assigning a handle to a user ID/session level. This handle would then be used to create a PSKT Database for communications between the SSS Child Process and APS Process pairs. The changes made in the new design can be found in Section G of Chapter IV.

The APS Processes also have read access to the database. An APS Process uses the database to determine the shared memory key of the PSKT Database that it will use for communications with the appropriate SSS Child Process that provides access to the network interface card.

Locking is used to ensure exclusive access to the database when a PSKT handle is allocated. This prevents the possible allocation of one PSKT handle to more than one SSS Child Process.

e. Pseudo-Socket (PSKT) Database

The Pseudo-Socket (PSKT) Database provides the communications interface between each SSS Child Process and APS Process pair. Entries in the database are dynamically updated during runtime. Each entry in the database includes a SSS Child Process identifier, APS Process identifier, network address of the client workstation, network address of the server, an *in use flag* field, a *APS status* field and two buffers. The *in use flag* marks entries in the database that are currently being used by a SSS Child Process/APS Process pair. The *APS status* field keeps track of whether the entry is currently initializing, done or ready. The two buffers are circular buffers used to pass the data stream between processes. One is used to pass the data stream from the server to the client and the other is used to pass the data stream from the client to the server.

The SSS Child Processes and APS Processes have read/write access to the database. When the SSS Child Process receives data from the client on a socket, the data is written into the client to server buffer. The APS Process then retrieves the data from the buffer for use by the APS. Output data from the APS is written into the server to client buffer. The SSS Child Process then reads the data in the buffer and sends the data to the client through the socket.

Locking is used to ensure exclusive access to the database during allocation of an entry in the database. This prevents the possible allocation of a PSKT to more than one SSS Child Process.

f. Database Initialization

The table below summarizes the existing databases and the processes that are responsible to initialize those databases. New databases introduced in the new design are summarized in Chapter IV. It also states which databases are statically configured and which use shared memory.

Database	**Process**	**Frequency**	**Statically Configured**	**Shared Memory**
Allowed Protocols	SSD	System-Wide (once on system startup)	Y	N
Allowed Trusted Path Extension (TPE)	TPS Parent	System-Wide	Y	N
PSKT	SSS Child	Per TPE User/ Session (every time a user/session is established)	N	Y
PSKT Map	SSD	System-Wide	N	N
User	TPS Parent	System-Wide	N	Y

Table 1. Database Initialization

Shared memory is used only for databases that are accessed by multiple processes. The two statically configured databases (Allowed Protocols Database and Allowed TPE Database) do not use shared memory. They are both read from files into memory by the process listed in the table. The PSKT Map Database is a runtime database which currently uses a shared flat file for implementation instead of shared memory. The PSKT Database and User Database are both runtime databases that use shared memory. The initialization of the Pseudo-Socket Map Database has been changed in the new design as discussed in Chapter IV.

3. Processes

a. Trusted Path Server (TPS) Parent Process

The Trusted Path Server (TPS) Parent Process runs during the startup of the MLS server as a system daemon. The TPS Parent Process determines whether each trusted path connection received from the TPE will be accepted. When the TPS Parent Process receives a trusted path connection, it checks for the TPE identifier in the Allowed

TPE Database. The TPS Parent Process forks a TPS Child Process to handle the connection if the Allowed TPE Database contains the TPE identifier. The TPS Parent Process must run at the level of the MLS LAN since it requires access to the network interface card.

b. TPS Child Process

The TPS Child Process takes over the trusted path connection once the TPS Parent Process has accepted the connection. The child handles all trusted path communications from the TPE including login, change session level, run and logout. When the TPS Child receives data from the client on the connection, the process first checks to ensure that the data is a Secure Attention Request Packet (SARP). The TPS Child then reads and processes the command. The output is sent back to the client on the connection. The TPS Child requires the privilege to access system identification and authentication information and uses the User Identification and Authentication Module to verify a username and password. The TPS Child Process also runs at the level of the MLS LAN. The TPS Child Process maintains the information in the User Database by associating each logged in TPE identifier with the user ID/session level.

c. Secure Session Daemon (SSD)

The Secure Session Daemon (SSD) is another process that runs during startup of the MLS server as a system daemon. The SSD reads in the static Allowed Protocols Database from a file and uses the execution path in each entry to start a SSS Parent Process for each protocol found in the database. The SSD must run at the level of the MLS LAN since the child it creates must run at that level in order to access the MLS LAN.

In the pre-existing design, the SSD requires privileges to: 1) read/write all secret and integrity levels, 2) bypass discretionary access checking and 3) change its user identifier. These privileges allow the SSD to clean up any left over PSKT Databases that could be at any user/session level. However, this function has been moved to the TPS Parent Process as described in Chapter IV.

d. Secure Session Server (SSS) Parent Process

The Secure Session Server (SSS) Parent Process validates any requests received from clients for services offered by APSs. When the SSS Parent Process receives a request, it first determines if the request was received from a valid TPE by verifying that the TPE is in the User Database. In the new design described in Chapter IV, the SSS Parent also checks for a valid remote connection in the Remote Connection Database if the TPE cannot be found. If a valid session or valid remote connection is found, the SSS Parent process forks a SSS Child Process to handle additional requests on the connection. Otherwise, the connection is terminated.

The SSS Parent Process must run at the level of the MLS LAN since it requires access to the network interface card. The SSS Parent Process requires the XTS-400 privilege to change the user identifier to the network user identifier to access restricted ports.

e. SSS Child Process

The Secure Session Server (SSS) Child Process creates an APS Process to service the client. The SSS Child Process then passes data between the client and the APS. Any data received on the socket is passed to the APS through using the PSKT Database. Any data placed in the PSKT Database by the APS is sent out on the socket by the SSS Child.

The SSS Child Process also runs at the level of the MLS LAN, and requires the MAC/DAC exemption and user identifier change privileges. The SSS Child Process normally runs at the level of the MLS LAN to access the network interface card, but it also needs to be able to access the PSKT Database being used for communications with the APS. In order to do this, the SSS Child Process must be exempt from MAC/DAC restrictions. The SSS Child Process also initializes the PSKT Database. This database is created at a specific user ID/session level which requires a change in user identifier in addition to the MAC/DAC exemption.

f. Application Protocol Server (APS)

Each Application Protocol Server (APS) handles the server side of a client/server protocol. These servers are untrusted and must use the PSKT interfaces to communicate with the socket connected to the client via the SSS Child Process. The source code of each server needs to be modified to use the PSKT interface. The handle needed to attach to the shared memory is retrieved from the PSKT Map Database.

A number of APSs have already been prototyped in other works. These include an Internet Message Access Protocol (IMAP) mail server [8] which was examined again [9], a Simple Mail Transfer Protocol (SMTP) mail server, Sendmail [10] and a Hypertext Transfer Protocol (HTTP) web server [11].

4. Other Modules

a. Semaphore

The Semaphore Module provides interfaces used to lock various databases. When a database is initialized, a semaphore is created for the module that manages the database. The semaphore identifier is opened by each additional process that accesses the same database. The PSKT Map Module and the PSKT Module depend on this module. The following new modules described in Chapter IV also depend on this module for locking: Remote Connection Module, Remote Pseudo-Socket (RAPSKT) Map Module, RAPSKT Module and Cleanup Module.

b. User Identification and Authentication

The User Identification and Authentication Module manages access to user identification and authentication functions provided by the XTS-400. The two interfaces included are used by the TPS Child Processes. The first allows the TPS Child Process to determine if a username and password sent via the TPE are valid. The second allows the TPS Child to determine if the username/session level pair is valid.

c. Buffer IO

The Buffer IO Module is used to manage a circular buffer. The PSKT Module depends on this module to manage the two buffers included in each entry of the database.

d. Privileges

The Privileges Module manages access to privilege granting and revoking functions provided by the XTS-400. Only privileges that are predetermined can be enabled by each process. Interfaces are provided to disable all privileges previously granted and to enable each of the following privileges:

- Read/write all secrecy and integrity levels and bypass discretionary access checking (MAC/DAC exemption)
- Perform identification and authentication checking
- Change owner/group attributes of the current process

- Change access class of an object
- Change mandatory access of an object
- Read objects with lower integrity

The SSD, the SSS Parent and Child, the TPS Parent and Child, and the User Identification and Authentication Module depend on the Privileges Module. The Trusted Remote Session Server (TRSS) Parent and Child described in Chapter IV also use the module.

e. Shared Memory

The Shared Memory Module provides interfaces that simplify the use of shared memory management functions provided by the XTS-400. The functions in the module allow a process to create, attach to, detach from and remove shared memory blocks. When a process attaches to a database in shared memory, it uses the shared memory key associated with the database to retrieve a pointer to a database in shared memory that has already been initialized. A process detaches to release the shared memory resources which are still being used by other processes. The User Module and PSKT Module depend on the Shared Memory Module. The following new modules described in Chapter IV also depend on this module: Remote Connection Module, Remote Pseudo-Socket (RAPSKT) Map Module, RAPSKT Module, Cleanup Module, Peer Level Module and Source Address Binding Module.

f. Utility

The Utility Module contains a variety of utility functions used for retrieving the access class of a process, comparing two access classes to determine if a read or write is allowed, evaluating a test condition (similar to ‘assert’ in UNIX) and debugging. All modules depend on this module except the Utility Module.

In the next chapter, the requirements for remote application support are discussed and the functional requirement identified.

III. CONCEPT OF OPERATIONS AND REQUIREMENTS

A. INTRODUCTION

The Monterey Security Architecture (MYSEA) server described in Chapter II currently does not have support to allow MYSEA thin clients to run remote applications on the multilevel secure (MLS) server over the MLS LAN. The goal of the new design is to add the necessary secure services to provide this remote application support. The remainder of this chapter discusses the concept of operations for the user and the requirements, constraints and dependencies that were considered during development of the design specification.

B. CONCEPT OF OPERATIONS

When the user desires to use an application with remote application (RA) support on the MLS server, the user must first invoke the secure attention key on the Trusted Path Extension (TPE) device, which establishes a secure connection with the server. The TPE is then used to actually login to the MLS server by entering a user name and password in response to prompts issued by the server. Still using the TPE, the user then chooses an initial session level. The session is finally started by issuing the *run* command on the TPE device. After the session-level connection between the MLS server and TPE is established, the client system is permitted to send and receive information from the server via the TPE. Now that the user has access to the MLS server, a web browser may be opened up on the client. A request is sent to the MLS web server to get the web page containing a list of RAs that can be run by the client. One of the links displayed in the web browser can then be chosen to start a specific RA.

C. TOP-LEVEL USER REQUIREMENTS

The design of the remote application (RA) support is based on the following top-level user requirements:

- The RA shall be able to communicate with the local MLS server, a remote MLS server and a RA server.
- The remote application shall be appropriately bound to the authenticated user's session.
- The user shall be able to launch the RA from the client.

- The MLS server shall be able to support both Unix/Linux and Microsoft Windows clients.
- The design shall only require a minimal number of changes to the RA.

D. OPERATIONAL CONSTRAINTS

The design of the RA support places the following constraints on the user operations:

- The user must use a uniform booting and login sequence as described above in Section B. This sequence is necessary for the RA mechanism to be properly initialized.
- The RA can only be invoked via a web interface.

E. ASSUMPTIONS

The following assumptions apply to the design of the RA support:

- Applications on the MLS server have been ported to provide RA support.
- Remote applications can only access the MLS LAN via the RAPSKT interface provided by the MYSEA trusted code.

F. REQUIREMENTS OVERVIEW

1. Trusted Remote Session Server (TRSS)

The user logs in to the MLS server at a session level and any RAs that the user runs should also run at the same level. However, as discussed in Chapter II, the XTS-400 has privileged TCP/IP ports that can only be accessed by processes running with the network user identifier. In addition, a process also needs to run at the level of the MLS LAN in order to access the network interface card between the MYSEA server and clients. This creates the necessity for a new trusted process on the server which allows the RA to communicate with the network interface card and access the privileged ports. The name of this new process is the Trusted Remote Session Server (TRSS). The TRSS provides a customized programming interface to allow the RA to communicate through the TRSS with local MLS servers, remote MLS servers and single-level RA servers.

To minimize the number of changes necessary to the RA implementation, the customized communication interface needs to allow the RA to make function calls using the same parameters used by the normal socket library function calls. This new communications interface is called the remote application pseudo-socket (RAPSKT). The TRSS manages all remote application pseudo-sockets for a given TPE and ensures that the RAPSKT used by an RA is associated with the correct user ID/session level pair for the given TPE. The RAPSKT mechanism enables data to be passed back and forth between the RA Process and TRSS Process.

The TRSS also prevents the RA from making connections to peers that do not have the same session level as the RA. For each request, the TRSS needs to control access to peers by comparing the level of the peer with the level of the RA. This requires a static configuration table containing a list of authorized peers along with corresponding session levels. If the peer is another TPE, the TRSS consults the existing User Database instead of the static table.

The TRSS must assign each RA connection a unique identifier associated with a user ID/session level. This allows the SSS Parent to determine when an incoming connection request is coming from an RA.

The TPS Child Process needs to have a means of terminating the TRSS Parent Process when a user logs out or changes session levels. Therefore, a signal mechanism is necessary to allow asynchronous termination signals to be sent from the TPS Child Process to the TRSS Parent Process.

2. Remote Application (RA)

All remote applications must be modified to communicate with the clients via the RAPSKT interface. The RA must be able to notify the TRSS when the RA needs to make a socket library function call. In addition, the TRSS Process needs to be able to terminate the remote applications that are not closed before a user logs out of the system or changes session levels. Therefore, some type of signaling mechanism is necessary for the RA.

In addition, a signal mechanism is required for the RA to send signals to the TRSS. However, the RA cannot use the normal signal mechanism to communicate with the TRSS. The RA must run at the session level of the logged in TPE and, therefore, the RA is not authorized to send signals to the TRSS with the normal mechanism. An alternate signal mechanism, discussed in Chapter IV, must be designed to handle this

.

To satisfy the need for the MLS server to support both Unix/Linux and Microsoft Windows clients and allow the user to launch RAs from the client, a web browser can be used to present the user with a list of possible RAs. A Common Gateway Interface (CGI) can then be used to start the RA itself.

3. Modification to Existing Modules

In order to appropriately bind the remote application to the authenticated user's session, the RAPSKT needs to be associated with the user ID/session level as well as with the TRSS and RA. This allows the RA to find the TRSS that it needs to communicate with and to find the RAPSKT that will be used for the communication. This binding needs to take place after a user logs into the MLS server, thus, from the RAPSKT perspective, the TPS Child Process's involvement in the RA mechanism is RAPSKT Database management.

The TRSS needs to be started by an existing process. Since the TPS Child Process is created after a user logs into the system, it makes sense for the TPS Child Process to create the TRSS. The TRSS then handles remote application requests from the user.

The SSS Parent also now needs to be able to determine if a connection request is coming from a remote application or from a user logged into a TPE. This information allows the SSS Parent to correctly determine where packets need to go.

G. FUNCTIONAL REQUIREMENTS

The following lists the functional requirements that the TRSS, RA, TPS and SSS Processes are to provide for RA support on the MLS server:

Trusted Remote Session Server (TRSS):

- The TRSS shall provide access to the network interface card for RAs by performing socket library function calls.
- The TRSS shall implement a RAPSKT interface used for communications between the TRSS and RA.
- The TRSS shall control requests for connection to peers (i.e., multilevel servers, single level servers and TPEs) from the RA

.

- The TRSS shall assign unique identifiers for each remote connection made by the RA and associate it with the user ID/session level, and remove all outstanding remote connections after a user logs out or changes session levels.
- A signaling mechanism shall be available for the TRSS to send signals to the RA and receive signals from the TPS Child and RA.

Remote Application (RA):

- The network related functions of the RA shall be modified to use the RAPSKT interface only for the MLS LAN side.
- A Common Gateway Interface (CGI) shall be used to start each RA.
- A signaling mechanism shall be available for the RA to send signals to the TRSS and receive signals from the TRSS.

Additional Requirements:

- The TPS Child shall start the TRSS after a user logs in. The TPS Child shall initiate cleanup for the previous session level and start a new TRSS when the user changes session levels. The TPS Child shall initiate cleanup when a user logs out.
- The TPS Child shall assign a RAPSKT Database to be used by the TRSS and RAs created for a user ID/session level.
- The SSS Parent shall be able to verify connections from remote application and TPEs.
- The RA support shall be designed to function with the existing trusted code on the MLS server including the Secure Session Server (SSS), the Trusted Path Server (TPS) and all related modules summarized in Chapter II.

The next chapter will discuss the high-level design of the RA support that satisfies these requirements.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. REMOTE APPLICATION SUPPORT HIGH-LEVEL DESIGN

A. INTRODUCTION

This chapter discusses the design process, the high-level design and the reasoning behind the various design decisions. The discussion starts with a description of the overall design and the concept of the remote application pseudo-socket (RAPSKT) interface. The chapter continues with a description of the new database modules starting with database initialization, moving on to an overview of how the databases are used and finishing with a description of each individual database. The third section of the chapter describes the new process modules and the changes to the remote application (RA) needed to run a multilevel secure (MLS) server. The next part of the chapter contains details regarding which socket library function calls are supported in the current design. Finally, there is a discussion on modifications to pre-existing modules necessary to support this design.

B. METHODOLOGY

Before starting to write the detailed specification, it was first necessary to research some background information. This started with a review of the particulars of programming on the XTS-400 in order to determine which system function calls would need to be used in the specification. Some study was necessary to learn the system commands on the XTS-400 system as well as the details of writing and compiling programs on the system. Previous research related to the project, including the existing design specification for the code currently implemented, was also reviewed.

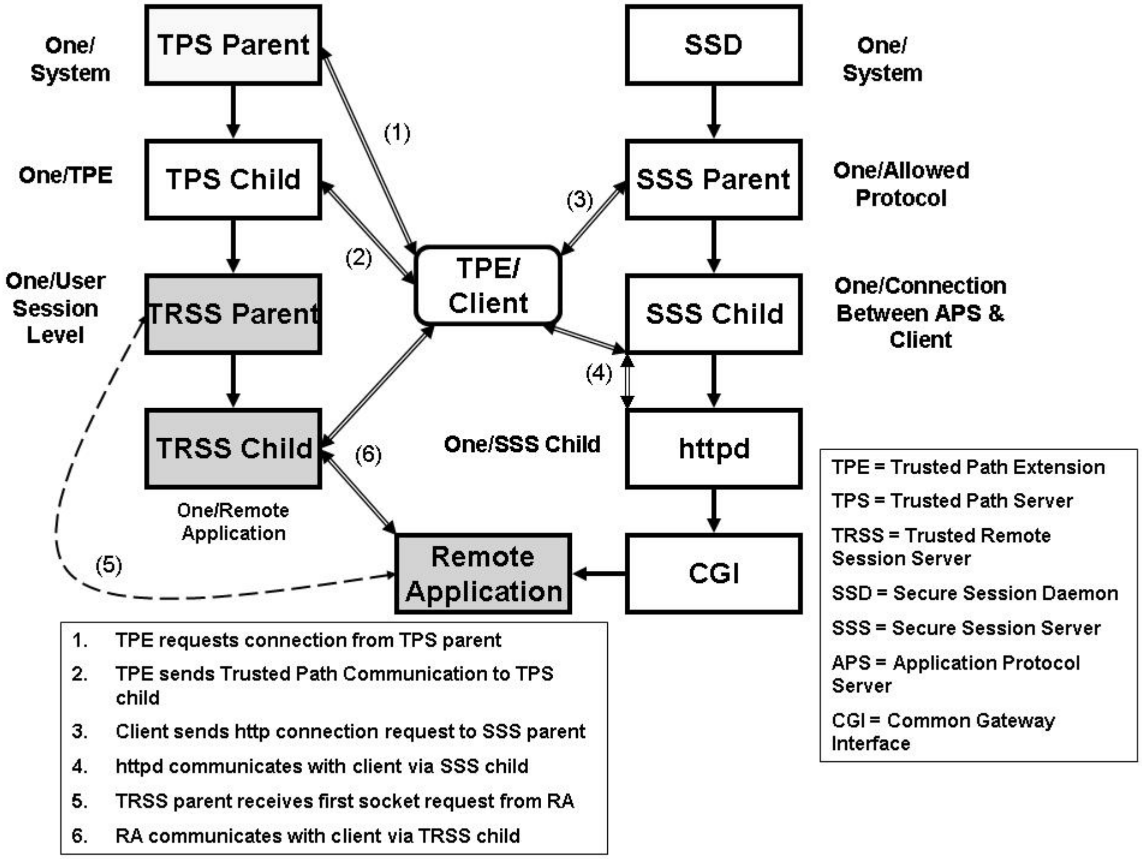
The design phase used an iterative process. The high-level design was first written and then several weekly meetings were used to review all high-level design decisions being made. The high-level design was then revised and the meetings continued until the high-level design was finalized.

Following completion of the high-level design, it was time to move on to the low-level design portion of the specification. Several modules were written at a time. After review by the design team, there were several meetings to go over any problems with the low-level design specification. This iterative process was continued through all modules.

Several times a fundamental problem was found with the high-level design, which made it necessary to first revise the high-level design before continuing.

C. DESIGN OVERVIEW

The figure below depicts the process structure for remote application (RA) support. New processes are shaded in gray.



1. TPE requests connection from TPS parent
2. TPE sends Trusted Path Communication to TPS child
3. Client sends http connection request to SSS parent
4. httpd communicates with client via SSS child
5. TRSS parent receives first socket request from RA
6. RA communicates with client via TRSS child

TPE = Trusted Path Extension
TPS = Trusted Path Server
TRSS = Trusted Remote Session Server
SSD = Secure Session Daemon
SSS = Secure Session Server
APS = Application Protocol Server
CGI = Common Gateway Interface

Figure 3. Design Overview

When the system first boots, the Trusted Path Server (TPS) Parent Process and Secure Session Daemon (SSD) are started. The SSD then starts a Secure Session Server (SSS) for each allowed protocol. The TPS Parent Process handles login requests from users and creates a TPS Child Process to handle trusted path commands from the users who log in or change session levels. At this point, the TPS Child Process also starts a Trusted Remote Session Server (TRSS) Parent Process to handle RA requests for a particular TPE.

After a user logs into the system, the user opens a web browser on the client to retrieve a web page with the links to start various RAs. The SSS Parent Process that handles HTTP requests creates a SSS Child Process for each connection. The SSS Child Process creates an HTTP daemon (httpd) to service the web request from the user. The httpd communicates with the client via the SSS Child Process, which can access the network interface card. A menu of remote applications is displayed in the web browser. After the user selects the RA to be started, a Common Gateway Interface (CGI) program starts the execution of the RA Process. The CGI program responds back to the client's web browser via the httpd with a 204 status code. This code tells the browser that the request succeeded, but that there is no new content. The SSS Child Process, httpd and CGI remain active and wait for additional requests until the connection is terminated or timeout is reached. The RA makes a request to the TRSS Parent Process for socket creation. Finally, the TRSS Parent Process creates a TRSS Child Process to allow the RA Process to communicate with the network interface card via the TRSS Child Process.

A Remote Application Pseudo-Socket (RAPSKT) mechanism allows communications to take place between the TRSS Child Processes and RA Processes. Before making a connection with the client, the RA must obtain a RAPSKT from the TRSS Parent. Subsequent socket calls are handled by the TRSS Child. When the RA needs to make a socket library function call using a socket, the function call type and any additional parameters for the function call are entered in the RAPSKT Database. The RA then waits for a response from the TRSS Child Process after signaling the TRSS Child Process that a function call needs to be handled. The TRSS Child wakes up and makes the socket library function call. Upon completion the return values are added to the RAPSKT Database. The TRSS Child Process then signals the RA that the function call is complete and goes back to sleep. The TRSS Child Process terminates when all sockets of the remote application are closed or a termination signal is received from the TRSS Parent Process.

D. NEW DATABASE MODULES

Six new databases have been added to the design to store the information needed by the various processes. These are the Remote Connection Database, the Remote Application Pseudo-Socket Map Database, the Remote Application Pseudo-Socket Database, the Peer Level Database, the Source Address Binding Database and the Cleanup Database.

The sections below contain high-level descriptions of the databases. Additional information including module interfaces, constants and the detailed design specification itself is included in the detailed design specification.

1. Database Initialization

The following table summarizes the location where initialization for each database takes place:

Database	**Process**	**Frequency**	**Statically Configured**	**Shared Memory**
Allowed Protocols	SSD	System-Wide (once on system startup)	Y	N
Allowed Trusted Path Extension (TPE)	TPS Parent	System-Wide	Y	N
Cleanup (*)	TPS Parent	System-Wide	N	Y
Peer Level (*)	TPS Parent	System-Wide	Y	Y
PSKT	SSS Child	Per TPE User/ Session (every time a user/session is established)	N	Y
PSKT Map	*TPS Parent*	System-Wide	N	N
RAPSKT (*)	TRSS Parent	Per TPE User/ Session	N	Y
RAPSKT Map (*)	TPS Parent	System-Wide	N	Y
Remote Connection (*)	TPS Parent	System-Wide	N	Y
Source Address Binding (*)	TPS Parent	System-Wide	Y	Y
User	TPS Parent	System-Wide	N	Y

(*) – New Databases (Italics) – Change From Pre-existing Design

Table 2. Database Initialization Summary

Two general rules were used to decide where to place the initialization of each of the new databases. Any system-wide databases not used solely by the SSD would be initialized in the TPS Parent Process. Any databases specific to a user ID/session level would be initialized in either the SSS Child Process for servers or TRSS Parent Process for RAs. The SSS Child Process and TRSS Parent Process both handle communications for only one user ID/session level. This changed the location of the initialization of the existing PSKT Map Database which originally took place within the SSD process.

Most databases are stored in shared memory. The shared memory keys used to find the shared memory segments are included in the detailed design specification. Statically configured databases are initialized based on administratively-defined data stored in read-only files. The contents of the Source Address Binding Database and Peer Level Database do not change during run-time. The contents of the remaining databases are updated during run-time.

The fact that shared memory is being used requires that the issue of locking be examined to prevent the possibility of processes reading inconsistent data. Locking will be discussed below for each individual database. The Semaphore Module discussed in Chapter II is used to implement the locking.

All databases include a variable which contains the current state of the database. This variable prevents processes from accessing the database until initialization is complete.

2. Database Overview

The figure below shows which databases each process accesses after the system has been initialized. The type of access for each process is also shown in the diagram. Additional details regarding specific accesses are provided in the sections that follow. The new processes and databases are shaded in grey.

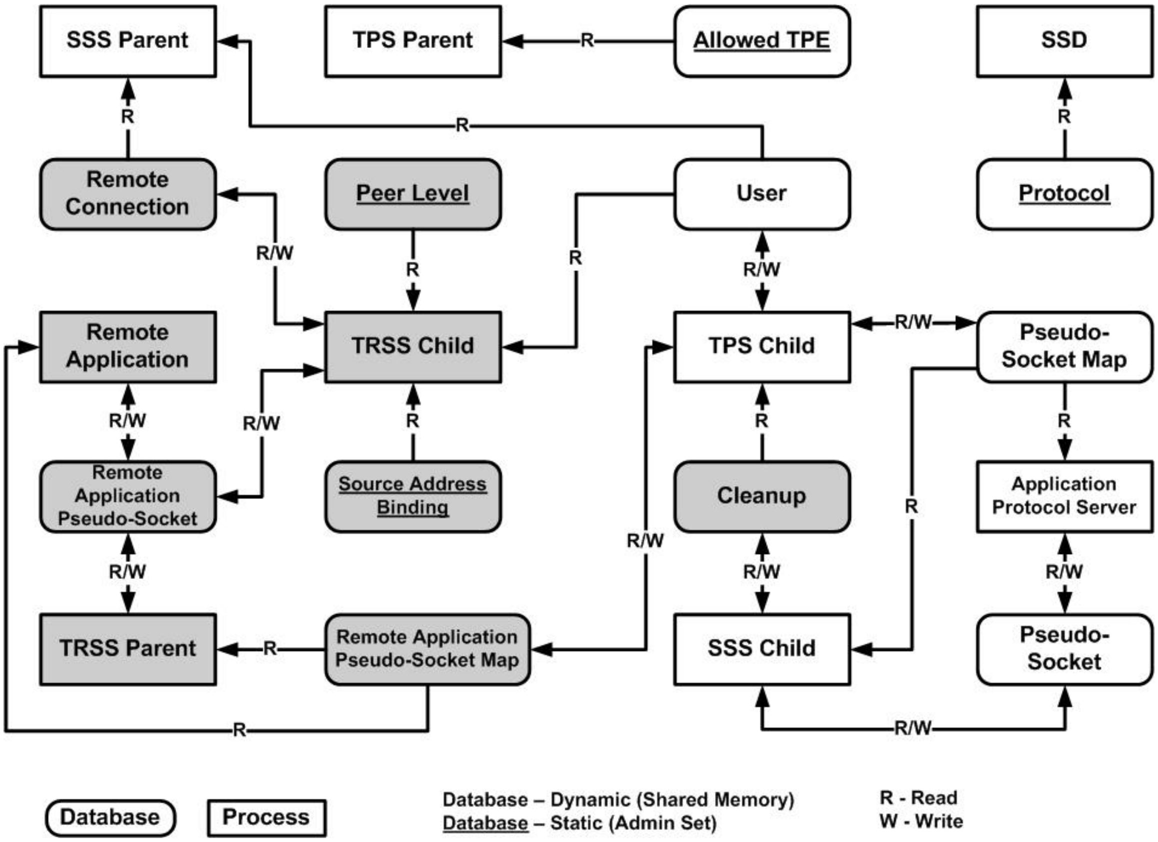


Figure 4. Process / Database Relationships

3. Remote Application Pseudo-Socket (RAPSKT) Map Database

The Remote Application Pseudo-Socket (RAPSKT) Map Database was added to keep track of the location of the shared memory segment used for communication between the RA Processes and TRSS Processes. The RAPSKT Map Database maps a user ID and session level to a shared memory key which is used to find the shared memory segment that contains a Remote Application Pseudo-Socket Database. The TRSS Parent Process identifier is also included with each entry.

Three processes use this database. The TPS Child Processes maintain the database by assigning a shared memory key to a user ID/session level and a TRSS Parent Process identifier when a user logs in or changes session level. When a user already logged into the system logs in again from another client at the same session level, the same shared memory key is used. The TRSS Parent Process and RA Process only have read access to the database to find which shared memory key they need to use. The

TRSS Parent Process identifier is used by the TRSS Parent to find the shared memory key for the RAPSKT Database described below. The user and session level fields allow the RA Process to find the same RAPSKT Database since the RA Process has no way of knowing the TRSS Parent Process identifier. The TRSS Parent Process identifier also allows the TPS Child Process to determine if a TRSS Parent Process has already been created for the requested user ID/session level.

The RAPSKT Map Database requires that a lock be obtained for both reading and writing. The lock prevents two TPS Child Processes from attempting to assign the same shared memory key to two different user ID/session levels. The lock prevents the possibility of either the TRSS Parent Process or the RA Process from reading inconsistent data if the TPS Child Process is rescheduled before it completes filling in an entry in the database.

4. Remote Application Pseudo-Socket (RAPSKT) Database

The RAPSKT Database is included in the design to allow communications between the RA Processes and the TRSS Processes. One instance of the database exists for each logged in user ID/session level. Each entry contains the following fields: call type, parameter and data buffer, variable length buffer flag, function return value, error number, in use flag, RA Process identifier, TRSS Child Process identifier and RAPSKT file descriptor. The *call type* field is used to specify the type of socket library function call being made by the RA. This will be described in greater detail in Section F. The two buffers are used to pass the parameters for the socket library function call being made. In some cases, the size of the parameter is too large to fit in the buffer included in the database so a dynamically allocated buffer is created to store the data. The variable *buffer flag* is turned on if the variable length buffer is used. The *function return value* and *error number* fields are used by the TRSS Child Process to return the return value for a function and the error number resulting from an error respectively. The *in use flag* field is used to mark RAPSKT Database entries that are currently being used by RAs. The *remote application identifier* and *TRSS Child Process identifier* fields are used to store the process identifier of the processes using the entry for communications. Finally, the identifying number for the RAPSKT associated with the socket library function call being made is stored in the *RAPSKT file descriptor* field. A separate data structure within the database is used to manage the RAPSKT file descriptors.

The TRSS Parent Processes, TRSS Child Processes and RA Processes all have read and write access to the database. The RA Processes use the database to make various socket library function calls using the RAPSKT by passing the function call type and associated parameters. Additional information on the supported functions is described in another Section F. The RA Process identifier field allows the RA to find the correct entry to use for communications. The TRSS Parent Process uses the database to create a TRSS Child Process for RA making a request for the first time. The TRSS Child Process uses the database to get the parameter for a socket library function call being made by a RA Process. It also uses the database to return the function return value and error number if necessary. The TRSS Child Process identifier field is used by the TRSS Child Process to find the entry being used for communications with the RA.

The RAPSKT Database only requires a locking mechanism for allocation and when a RAPSKT is closed or shutdown. During allocation the lock prevents multiple RA Processes and TRSS Child Processes from allocating the same RAPSKT. The lock for closing or shutting down a RAPSKT is necessary because of the way RAPSKT file descriptors are handled. The possibility exists after a *fork* of the TRSS Child Process that the same RAPSKT file descriptor could be used by more than one TRSS Child Process/RA Process pair. Thus it is necessary to ensure that no processes are still using the RAPSKT file descriptor before freeing it in the RAPSKT file descriptor array. Otherwise, the database requires no additional locking because the database entries used by a TRSS Child Process/RA Process pair are not used by any other processes, and the TRSS Child Process and RA Process are in lock step. The TRSS Child Process and RA Process signal the other process and wait for a response before continuing their work.

5. Remote Connection Database

The Remote Connection Database is used to bind the remote application’s connection identifier to a user ID/session level. This connection identifier consists of the source IP address, source port number, destination IP address and destination port number. This binding is necessary for the SSS Parent Process to determine if the connection request came from a RA. Otherwise, the SSS Parent will just drop the packet since it only looks at this database after it checks the User Database.

Both the TRSS Child Process and SSS Parent Process use the database. The TRSS Child Process adds entries to the database when a RA makes a network connection and removes entries from the database when the connection is closed. The SSS Parent Process only has read access and uses the database to check the user ID/session level of a remote connection when the connection is not made by a user logged in from a TPE.

A lock needs to be obtained for both reading from and writing to the Remote Connection Database. Since the database is used to check the user ID/session level of a remote connection, the lock is necessary to prevent the possibility of the SSS Parent Process reading inconsistent data if the TRSS Child Process does not complete an entry in the database before being rescheduled. It also prevents a remote connection from not being found that is in the process of being added to the database or being found that is in the process of being removed from the database.

6. Peer Level Database

The Peer Level Database is used to keep track of the security level of the peer system with which the RA communicates. A peer system can be either an MLS system (e.g., another MYSEA server) or a single-level host. For single level systems, a field indicating the level of the system is also included. This database provides the means to identify whether or not a process will be allowed to communicate with a peer based on the level of the RA Process and the level of the peer system. This information is necessary to stop a RA from making a remote connection to a peer at a level not allowed by the security policy. Only the TRSS Child Process requires read access to the database. The IP address of the peer is used to retrieve the type of system (single or multi level) and level of single level systems.

The Peer Level Database does not need an additional locking mechanism at this point beyond the variable that is used to indicate that database initialization is complete. The information in the database comes from a static file and does not change during runtime. If the database is changed to allow dynamic updating, the locking issue will have to be re-examined. Suggestions for dynamic updates of this database are described in Chapter V.

7. Source Address Binding Database

The Source Address Binding Database was the last database module added to the design after an additional problem was found in the detailed design phase. Some calls such as *connect* normally bind the socket to any address before the socket library function call is made. However, for the case in which the RA's peer is another MYSEA server, the remote connection identifier needs to be added to the Remote Connection Database so that the SSS Parent Process can verify a connection request (resulting from the bind operation) if it is not coming from a user logged in with a TPE. The socket must be bound before the remote connection identifier can be added to the Remote Connection Database. Therefore, a method was needed for the TRSS Child to determine which source IP address to use for each destination IP address so that the socket can be appropriately bound. This is accomplished in the Source Address Binding Database. The TRSS Child Process has read access to the Source Address Binding Database. A destination IP address, to which a network mask is applied, is used to retrieve the source IP address.

The Source Address Binding Database does not need an additional locking mechanism since the information in the database comes from a static file and does not change during runtime. If the database were to be changed to allow dynamic updating, a locking mechanism would be required. Suggestions for dynamic update of this database are described in Chapter V.

8. Cleanup Database

The Cleanup Database is used to keep track of the active SSS Child Processes in the system. This is necessary because the TPS Child Process needs to initiate cleanup when a user logs out or changes session levels. The TPS Child Process already knows the process identifier of the TRSS Parent Process that must be killed, but it needs to have a way to get a list of SSS Child Processes that are currently running on the system. Each SSS Child Process is associated with a TPE identifier in the Cleanup Database so the TPS Child Process can kill all processes associated with a user ID/session level. The SSS

Child Processes add their process identifiers to the database when they are created and remove their process identifier when they are terminated.

The Cleanup Database requires a locking mechanism for both reading and writing. It is important that TPS Child Processes do not read any inconsistent data from the database during cleanup since the SSS Child Process identifiers found for the TPE are signaled to terminate.

There is one possible issue with SSS Child Process cleanup which requires more work. In the unlikely occurrence that an SSS Child Process has just been created and the TPE logs out or changes session levels before the SSS Child Process identifier can be added to the Cleanup Database, the SSS Child Process currently will not be terminated. This could potentially cause a resource exhaustion condition to occur due to the presence of too many “zombie” processes.

E. NEW PROCESS MODULES

Three new processes were added to the existing design: the TRSS Parent Process, the TRSS Child Process and the RA Process. The RA Process runs the RA program that has been modified (ported) to use the RAPSKT interface. This section provides the high-level details of these processes. Further information can be found in the detailed design specification.

1. Trusted Remote Session Server (TRSS)

The TRSS Processes are responsible for handling requests made by the RA Processes. The work is divided between a parent process and multiple child processes.

a. TRSS Parent Process

The TRSS Parent Process handles socket requests from new RAs for each user at a specific session level. The following flow diagram shows the high-level flow for the process:

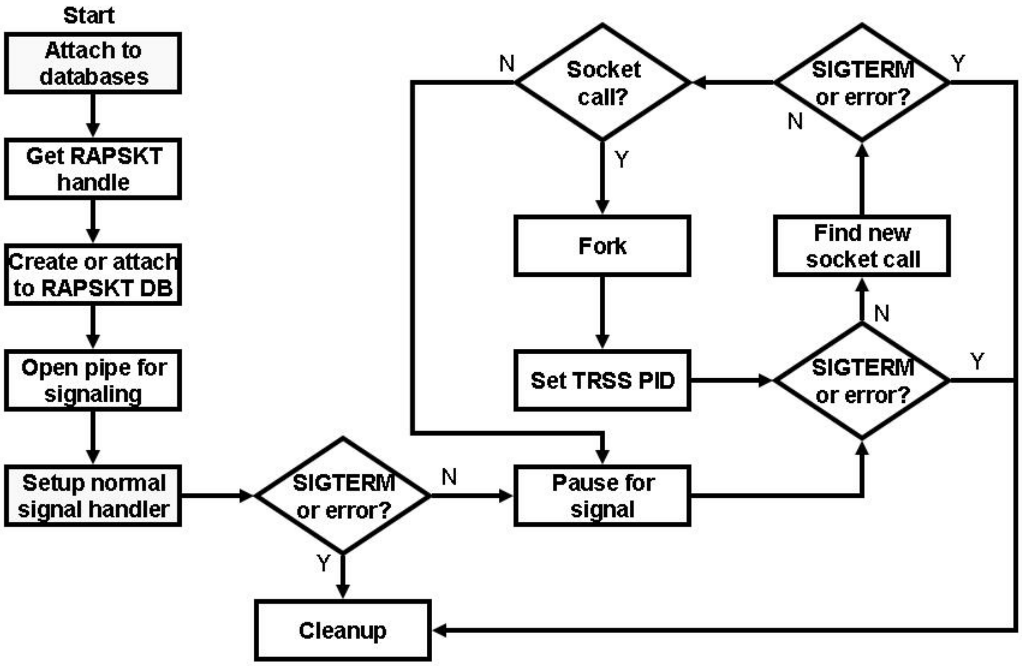


Figure 5. TRSS Parent Process Flow Diagram

As shown in the diagram, the TRSS Parent Process first attaches to all databases that the process needs access to. The process then gets the shared memory key that will be used for the RAPSKT Database from the RAPSKT Map Database. The process uses this key to create and initialize the new RAPSKT Database. If the user ID/session level has already logged into the system from another TPE, the TRSS Parent Process will instead attach to a RAPSKT Database which has already been created for the other TPE. Next a named pipe is created to use for signaling for reasons discussed below. A normal signal handler is also setup to capture normal signals from the TPS Child Process. The process then pauses for requests from RA Processes.

When the TRSS Parent gets a request to create a socket for a new remote connection from the RA Process, the TRSS Parent Process locates a RAPSKT database entry with the *call type* set to *socket* and creates a child process to handle the work of the RA. The TRSS Parent Process then enters the process identifier of the forked process into the RAPSKT Database. Since the TRSS Parent handles only the first *socket* function call and the TRSS Child Processes conducts all additional *socket* function calls, there needs to be a method for the TRSS Parent Process to differentiate which *socket* function calls in the RAPSKT Database it needs to handle. Therefore, two different *socket* call type variables were created for this purpose. After handling the first *socket* call found, the TRSS Parent Process will continue to search the remaining entries in the RAPSKT Database and will handle any additional *socket* calls found.

The TRSS Parent Process runs at the level of the MLS LAN since the children it creates must be forked at that level. The XTS-400 requires that all processes using the MLS LAN network interface card run at this level. The normal *kill* signaling function call cannot be used by the RA to signal the TRSS Parent since the RA is not authorized to signal another process running at a different level. Therefore, an alternate signaling mechanism is needed to avoid having to use polling, which could make the processes less efficient. The chosen method uses named pipes and is described below in Section 3 (Synchronization).

The TRSS Parent Process also needs to have a signal handler to capture signals sent using the *kill* function call. This handler is necessary to allow the TPS Child Process to send a terminate signal to the TRSS Parent Process when a user logs out or changes session levels. When the signal is captured, a flag is set in TRSS Parent module data by the signal handler. If the process was paused when the signal was received, the flag is checked immediately after the process is woken up. The flag is also checked after the TRSS Parent Process attempts to find another socket call in the RAPSKT Database and after the work for each socket call is complete. This causes the main processing loop of the TRSS Parent Process to exit and cleanup of the module to begin.

The TRSS Parent Process requires the MAC/DAC exemption and user identifier change privileges. The MAC/DAC exemption privilege is necessary to allow a process running at one level to access data at another level. In the case of the TRSS, both TRSS Parent and TRSS Child Processes, running at the level of the MLS LAN, must be exempt from MAC/DAC restrictions so that they can access the RAPSKT Database, which is associated with a specific user/session level. The TRSS Parent Process also initializes the RAPSKT Database. This requires a change in user identifier in addition to the MAC/DAC exemption since the TRSS Parent Process is creating the RAPSKT Database in shared memory.

b. TRSS Child Process

The TRSS Child Process handles all the work for socket library function call requests made by the RA Process. The diagram below shows the high-level process flow.

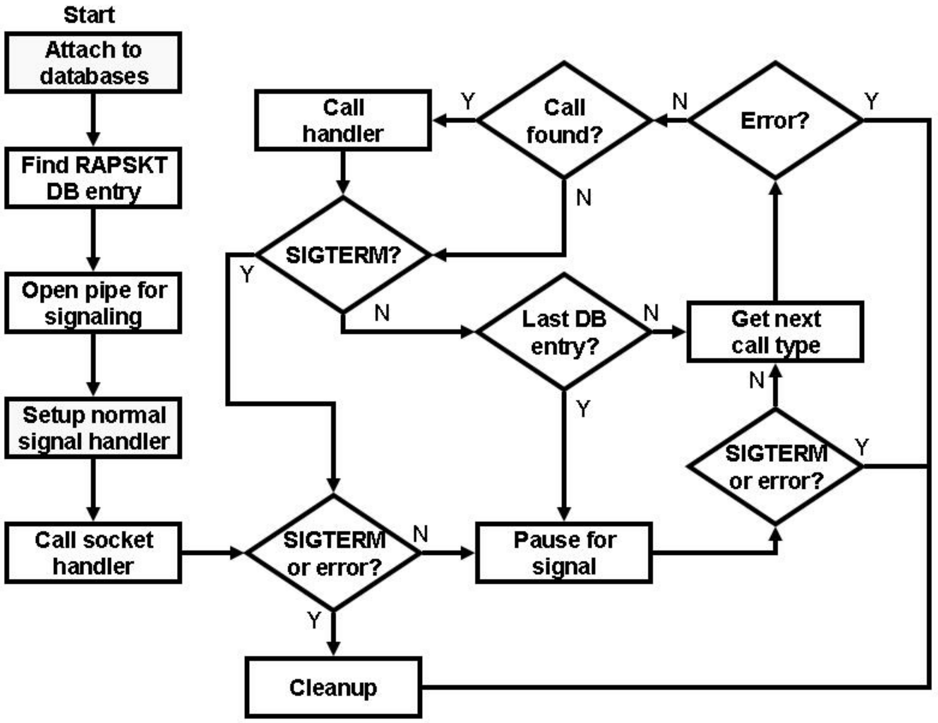


Figure 6. TRSS Child Process Flow Diagram

When the process initially starts, the process attaches to the necessary databases and creates the socket for the RA Process. The TRSS Child Process then finds the entry used in the RAPSKT Database for the initial *socket* call. Once the entry is found, a named pipe and signal handler are setup as discussed in the TRSS Parent Process section above. The process then waits for a signal from the RA Process. When the TRSS Child Process receives a signal from the RA, the TRSS Child Process checks the call type for each entry in the database. For any entry found with a valid socket library function call, the TRSS Child Process calls the appropriate function handler.

The TRSS Child Process also must run at the level of the MLS LAN in order to allow communications with the TPE. However, this creates a signaling problem between the remote application process and TRSS Child Process. The same signaling mechanism used in the TRSS Parent Process is used for the RA to signal the TRSS Child Process. However, when the TRSS Child Process needs to send a signal back to the RA, the normal signaling mechanism can be used because the TRSS Process is authorized to signal other processes running at different levels.

The TRSS Child Process also needs a signal handler to capture termination signals. It functions in the same way as the signal handler in the TRSS Parent Process. The flag set in TRSS Child module data is checked immediately after the process wakes up from pausing and after each socket library function call is handled.

The TRSS Child Process also needs the MAC/DAC exemption privilege for the same reason as the TRSS Parent Process.

2. Remote Application (RA)

Each client application that runs on the MLS server must have the source code modified to use the RAPSKT interface. Code must be added so that the shared memory key for the RAPSKT Database is first obtained from the RAPSKT Map Database so that the RA Process can attach to the shared memory. All network related function calls and the *fork* function call must also be changed to use the interface provided in the RAPSKT Module.

3. Synchronization

When the TPS Child Process creates a TRSS Parent Process, the new process first needs to retrieve the shared memory key that will be used for its RAPSKT Database. It is possible that the TRSS Parent Process may try to retrieve this value before the TPS Child Process has a chance to fill in the field in the RAPSKT database. Therefore, the TRSS Parent Process will need to loop until its process identifier can be found in the database or until a timeout is reached causing an error.

The RA is in lock step with the TRSS Child and Parent Processes. Whenever the RA wants to create its first RAPSKT, the RA Process signals the TRSS Parent Process before pausing. The TRSS Parent Process then creates a TRSS Child Process to handle the actual *socket* function call. When the RA wants to make some other socket library function calls including subsequent *socket* calls, the RA Process signals the TRSS Child Process before pausing. The RA then waits for a response from the TRSS Processes before continuing. When the TRSS Child Process completes the socket library function call, it signals the RA.

A separate named pipe mechanism is used to send signals from the RA to each TRSS Parent Process or TRSS Child Process. When the TRSS Parent Process starts, the TRSS Parent creates a named pipe with a file name appended with its process identifier. The TRSS Parent Process then pauses by calling *select* on the file descriptor of the named pipe, waiting for a RA Process to write to the file. When the RA Process first accesses the RAPSKT Database, the RA looks up the named pipe with the TRSS Parent Process identifier in the RAPSKT Map Database for a specific user ID/session level and opens it. When the RA needs to create its first socket, the RA writes to the named pipe to signal the TRSS Parent Process which spawns a TRSS Child Process. This TRSS Child Process creates another named pipe with a file name appended with its process identifier. The RA uses this new named pipe to send any further signals to the TRSS Child Process. The TRSS Child Process uses the normal signal mechanism to signal the RA and uses the *select* call on the file to wait for signals from the RA.

F. SUPPORTED SOCKET FUNCTIONS

The table on the next page contains a list of socket library function calls that can be made by the RA Process. In addition there is also a *pseudo-fork* function (rapskt_fork). This is necessary to emulate the socket property of the *fork* function call. Otherwise, the RAPSKT used for communications would not be copied to the child process.

There were two choices in the design regarding where to put the actual socket calls and policy checking for the various functions. All of the work could be placed in the RAPSKT Module with the TRSS Processes doing nothing except checking for the type of call and calling the appropriate handler function in the RAPSKT Module. The other possibility was to make the RAPSKT Module a means to pass the function parameters to the TRSS Processes and then let handlers in the process module do the actual socket call and policy checking. The second option was selected because no process other than the TRSS Child Process needs to know about the actual socket since only the TRSS Child Process can communicate directly with the network interface card.

Socket Call	**RAPSKT Call**	**Limitation**
accept	rapskt_accept	none
bind	rapskt_bind	none
close	rapskt_close	none
connect	rapskt_connect	none
fcntl	rapskt_fcntl	Limited command types
getpeername	rapskt_getpeername	none
getsockname	rapskt_getsockname	none
getsockopt	rapskt_getsockopt	none
ioctl	rapskt_ioctl	Limited command types.
listen	rapskt_listen	none
read	rapskt_read	none
recv	rapskt_recv	none
recvfrom	rapskt_recvfrom	none
select	rapskt_select	Only socket file descriptors within same RA
setsockopt	rapskt_setsockopt	none
send	rapskt_send	none
sendto	rapskt_sendto	none
shutdown	rapskt_shutdown	none
socket	rapskt_socket	none
write	rapskt_write	none

Table 3. Supported Socket Functions

The remainder of this section describes the important aspects of the various socket library function calls. A complete description can be found in the detailed design specification.

1. Common Characteristics

All of the handlers within the TRSS Child Process share a few common steps. The function must first retrieve the function parameters from the RAPSKT Database using the appropriate database. The actual socket file descriptor must then be retrieved from module data for all calls except for *fork* and *socket*. Next the actual socket library function call can be made. The handler then writes the function return value and any other return values into the RAPSKT Database plus the error number, if the function returned an error. Any return values that are returned in the data buffer do not need to be copied since a pointer to the buffer is passed back from the get parameter function. Finally, the TRSS Child Process signals the RA Process before pausing for another signal. However, some of these functions require additional steps and checks.

2. Peer Level Checks

Any socket library function calls that access a peer must first check the level of the peer before allowing the socket library function call to proceed. The session level of the RA Process is compared to the value retrieved from the Peer Level Database. This check is necessary for *accept*, *connect*, *recvfrom* and *sendto*. If the peer's IP address is not included in the Peer Level Database, the handler checks to see whether the peer happens to be a logged in TPE and then verifies the session level if it is a logged in TPE.

3. Explicit Binding

The socket library function calls, which normally would bind the socket to a wild card address, if the socket has not already been bound, must first be bound to an address retrieved from the Source Address Binding Database. The reasoning behind this has already been explained in the Source Address Binding Database Section D above. The socket library function calls *connect* and *sendto* both require this binding.

4. RAPSKT Database Allocation and Deallocation

Several of the socket library function calls perform allocation or deallocation of RAPSKT Database entries and file descriptors. The *accept* and *socket* function calls need to allocate a RAPSKT file descriptor and a new entry in the database. The *close* and *shutdown* function calls do the exact opposite; they deallocate the database entry and RAPSKT file descriptor. The handler function in the TRSS Child takes care of the allocation for the *accept* call and the RAPSKT function call made by the RA Process takes care of the allocation for the *socket* call. The deallocation for the *close* and *shutdown* function calls also takes place in the RAPSKT Module. This is necessary because the database entry needs to be cleared, but this cannot be done before the return value can be retrieved from the RAPSKT Database by the RA. The *fork* function call also needs to allocate additional RAPSKT Database entries, but this will be discussed below separately.

5. Remote Connection Updates

Any socket library function call that either sets up or closes a remote connection needs to update the Remote Connection Database. Both the *connect* and *sendto* function calls add new entries to the database. The *close* and *shutdown* function calls remove entries from the database.

There is one possible issue remaining which has not yet been solved for *sendto*. The *sendto* function is normally used for connectionless protocols which do not use *close* or *shutdown* functions. Therefore, the TRSS Child Process will not know when to remove the remote connection identifiers associated with the *sendto* function from the database. When the TRSS Child Process terminates, part of the cleanup process involves removing all remote connection identifiers that are stored in the TRSS Child's module data. However, during runtime the TRSS Child Process will not know when the SSS Parent Process is finished with the remote connection identifiers, for connectionless protocols. Hence, it is possible that the Remote Connection Database would be filled with "zombie" entries while a user is logged into the system. All Remote Connection Database entries related to a TRSS Child Process are removed from the database when the process terminates which occurs when a user logs out or changes session levels.

6. Select

The *select* function call requires special handling. The file descriptors passed as parameters are RAPSKT file descriptors and need to be converted before the actual socket library call can be made. After the actual socket call completes, the actual file descriptor must also be converted back to the RAPSKT file descriptor. The handler function in the TRSS Child Process performs these conversions.

This RAPSKT version of *select* can only be used on sockets and only on the sockets in control of one TRSS Child Process. The normal *select* function call can be used for other file descriptors besides sockets. Socket file descriptors and other file descriptors also cannot currently be combined into one function call, but this is not something that would normally be done in a network application.

7. Variable Parameters

The function calls *fcntl* and *ioctl* both have a third parameter which could be of variable types. The request parameter for *ioctl* or the command parameter for *fcntl* determines the variable type of the third parameter. Thus it is necessary to first check the command type before copying the third parameter in and out of the RAPSKT Database. For *fcntl* the third parameter may not even exist so it is also necessary to use a variable argument list for copying the parameter into the RAPSKT Database. The corresponding RAPSKT functions for both of these function calls do not currently implement all possible command types, but these could easily be added to the design as needed. The supported command types are included in the detailed design specification.

8. Fork

The function call *fork* is the one non-socket function call that needs to be included in the RAPSKT interfaces. The diagram below shows the sequence of events that occurs when the RA Process calls the rapskt_fork function.

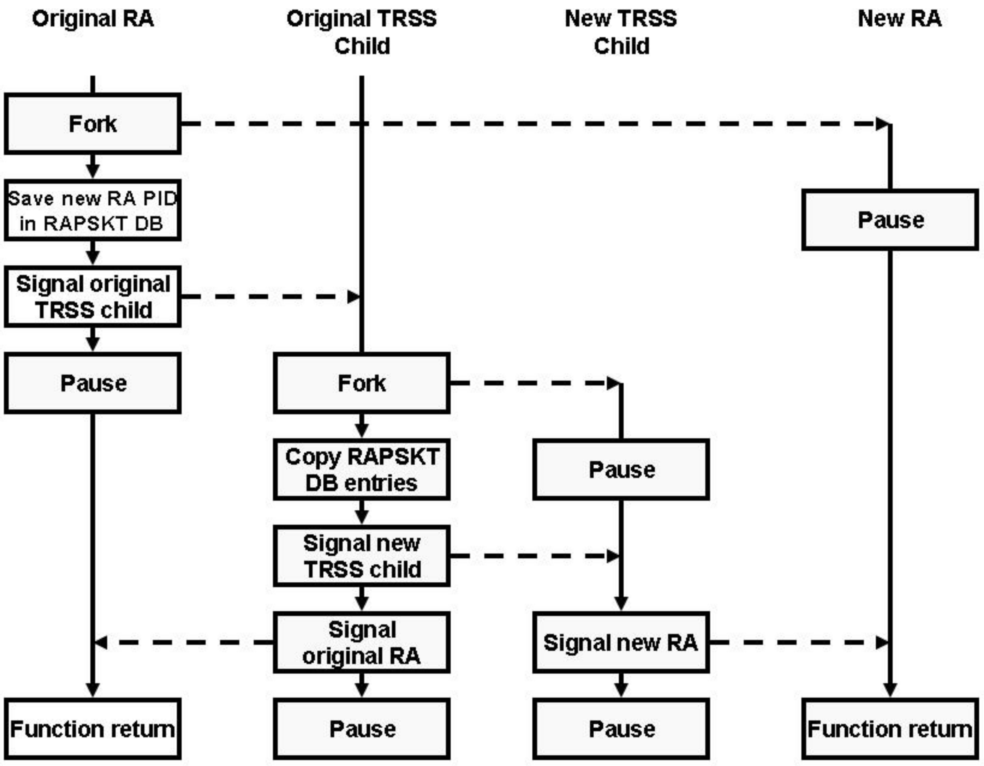


Figure 7. Fork Processing Sequence

This sequence is necessary in order to duplicate the RAPSKTs currently used in the original RA Process in the new RA Process. The RA Process forks first before signaling the TRSS Child Process to handle the duplication. The RA Process also saves the new process identifier in the RAPSKT Database so that the TRSS Child Process knows how to fill in the RA Process identifier field in the RAPSKT Database for new entries. When the TRSS Child Process receives the signal, it first forks a new TRSS Child Process which waits for a signal from the original TRSS Child Process. The original TRSS Child Process does the duplication by using an interface in the RAPSKT Module including setting the *RA Process identifier* and *TRSS Child Process identifier* fields to the values of the new forked processes. The original TRSS Child Process also sets the *function return value* field in the first RAPSKT Database entry for each RA/TRSS Child pair. Each TRSS Child Process then signals the RA Process that they will communicate with after the original TRSS Child Process completes the duplication and signals the new TRSS Child Process.

G. CHANGES TO EXISTING MODULES

The main changes to the existing modules are detailed below. In general, the cleanup processing within the TPS Child and Parent Processes, SSS Child and Parent Processes and SSD Process have been improved. Some additional work may still be required in this area. Further information is available in the detailed design specification.

1. TPS Parent Process / SSD

The initialization of the PSKT Map Database has been moved to the TPS Parent Process from the SSD. Another change is the enabling of the MAC/DAC exemption privilege. The TPS now cleans up any left over PSKT Databases and needs to be at the same level as each PSKT Database before this can be done. This clean up used to occur in the SSD in the pre-existing design.

2. TPS Child Process / SSS Child Process

Two main changes occurred in the TPS Child Process. Allocation of a PSKT handle is now the responsibility of the TPS Child Process, not the SSS Child Process. The TPS Child Process must now create a TRSS Parent Process to handle RA requests. Before creating the new process, the TPS Child Process first ensures that a TRSS Parent Process does not already exist for a user ID/session level.

3. SSS Parent Process

The SSS Parent checks the user ID/session level of a TPE when it receives a packet. Now when the TPE identifier cannot be found in the User Database, the SSS Parent will also try to find the remote connection identifier in the Remote Connection Database to get the user ID/session level.

4. User Database

One additional interface has been added to the User Module. This new interface allows the TPS Child Process to determine if a user ID/session level is logged in from any TPE. This interface is used during cleanup by the TPS Child Process. Since the same TRSS Child Process is used by all TPEs associated with the same user ID/session level, the TRSS Child Process is only terminated if a user ID/session level has logged out from all TPEs.

The TRSS Child Process now also has read access to the database. The TRSS Child uses the database to determine the security level of a peer that is not included in the Peer Level Database. This can occur if the peer is a logged in TPE.

H. SUMMARY

The top level design for RA support and the methodology used for the design have been presented. All databases and processes necessary to provide the support have been described. The socket library function calls supported have been presented and the changes to existing modules summarized. The next chapter will present possible future work and concluding remarks.

V. FUTURE WORK AND CONCLUSION

A. INTRODUCTION

This chapter discusses some possibilities for future work and presents the final thoughts on the project. A detailed design specification is a living document and there is always room for improvement. Several suggestions are given below for improvements to the design. A few ideas are also listed for testing the source code after programming is complete.

B. FUTURE WORK

1. Programming and Testing

The design specifications created for the remote application support contain sufficient detail that a prototype can be quickly implemented. The testing of this implementation is also challenging because of the complexity of the RAPSKT library.

a. Code Testing

The test plan for the source code should include both testing for the individual modules and testing for the overall remote application support. The latter can be achieved by porting a remote application. However, the means to provide remote application I/O support on the client are currently limited. Currently, only a web browser interface is available for the MYSEA user to access read-only data on the MYSEA server. Eventually a network-based windowing system, such as X Windows, could be ported to allow the remote applications to redirect its input and output to the MYSEA client. Meanwhile a simpler testing scheme should be used to test the remote application support.

Several testing ideas have been discussed as the design progressed. The Trivial File Transfer Protocol (TFTP) client program could be used for the test application. An open source TFTP client program would be ported to use the RAPSKT interface. The modifications to the TFTP client would allow the TFTP client to get a file from the TFTP server running on the MYSEA client. The user would start the TFTP client remotely from the MYSEA client by selecting the appropriate option presented by the CGI program in the web browser as discussed in Chapter IV. Unfortunately the TFTP client program does not exercise all of the socket function calls currently supported by the RAPSKT interface. Therefore, additional remote applications or other test code would be required to test all the RAPSKT functions.

b. Stress Testing

The Source Address Database, the Peer Level Database, the RAPSKT Map Database, the PSKT Map Database, the PSKT Database, the RAPSKT Database, the User Database, the Remote Connection Database and the Cleanup Database are all accessed by more than one process. Some testing may be required to determine if there are any bottlenecks in the system. There should not be any issues with the Source Address Binding Database and the Peer Level Database because no locking is required after initialization is complete, but the remaining databases could be potential problems because locking is required during runtime.

2. Design Improvements

Some design decisions were made to simplify implementation and testing of the initial prototype. This section contains some suggested changes which could be added to the project in future work. However, eventually it will be necessary to prove that the trusted code functions correctly. Adding more complexity to the design increases the difficulty doing this proof for the trusted code.

a. Configurable Databases

The database modules all currently have a static size which can be changed by modifying the system wide constants that define the maximum size of each database. However, in the future, the databases should be modified to allow the maximum size to be configured by an administrator. These databases should never be allowed to grow dynamically because of the complexity that would be added to the trusted code.

b. Source Address Binding Database/Peer Level Database

Both the Source Address Binding Database and Peer Level Database are loaded into shared memory during initialization from administratively set files. There is currently no way to modify these databases during runtime. An interface could be added to each module to allow a new version of the database to be loaded without having to reboot the system. Each database would need to be locked while being updated. In addition, the Peer Level Database could be modified to allow IP address ranges in order to better handle access to a range of single level hosts. Otherwise, there would need to be a large number of entries in the database to handle all allowed host addresses.

c. RAPSKT Map Database/PSKT Map Database

A covert channel storage currently exists in the RAPSKT Map Database and PSKT Map Database. The remote applications currently all have access to the same RAPSKT Map Database in order to determine which RAPSKT Database that they need to use for communications with the appropriate TRSS Child Process. The same applies for the PSKT Map Database that is used by the application protocol servers to determine which PSKT Database that the APS Processes need to use for communications with the appropriate SSS Child Process. One possible solution to mitigate this risk is to partition the collective map into multiple maps, one for each security level.

d. Cleanup

There are currently two potential cleanup problems. The problem concerning the existence of SSS Child "zombie" processes has already been discussed in Chapter IV. In addition the TRSS Child Process normally signals the remote application process to terminate, but it is possible that a user might be able to log out or change session levels before a TRSS Child Process can be created. Therefore, the remote application process would never receive the signal to terminate. Both of these events should be rare, if they ever occur, but pose a difficult problem that will take considerable design work.

e. Remote Connection Database

There is also an issue with "zombie" entries in the Remote Connection Database resulting from the fact that the TRSS Child Process will not know when to remove the remote connection identifiers associated with the *sendto* function from the database as discussed in Chapter IV. However, this problem has a limited effect since all Remote Connection Database entries related to a TRSS Child Process are removed from the database when a user logs out or changes session levels.

C. CONCLUSION

This thesis proposes a design for remote client-side application support on the MYSEA server. A new TRSS Process family has been proposed to accept remote connection requests for remote applications and perform the actual function calls related to the socket. A RAPSKT interface has been proposed to provide the communications interface between the TRSS Process and the remote application. The RAPSKT interface limits the number of changes required in the remote application. The number of new trusted processes has been limited by placing all new trusted code necessary for remote application support within the TRSS Process. This allows the remote applications to remain untrusted.

The design has been divided into well-defined modules. The RAPSKT Module manages the RAPSKT interface. The Remote Connection Module keeps track of the current socket connections for remote applications. The RAPSKT Map Module keeps track of the RAPSKT Database shared memory keys allocated to different user/session level. The data for peer IP addresses and their associated session level are managed by the Peer Level Module. The Source Address Binding Module manages the data that determines which source IP address to use for each destination IP address for sockets that need to be specifically bound. The TRSS Module accepts and handles remote connections requests from the remote applications.

This remote application support enhances the usability of the MYSEA MLS network by providing the much needed capability for MYSEA clients to run remote applications on the MYSEA MLS server over the MYSEA MLS LAN. This support provides the MYSEA user the ability to use server-resident client-side applications on the MYSEA server to access data at different security levels.

Continued research on the MYSEA project could have far reaching benefits for future military and Department of Defense (DoD) transformation initiatives that require a high-level of information assurance. This could include initiatives such as the Global Information Grid (GIG), FORCEnet or Homeland Defense.

APPENDIX DESIGN SPECIFICATION

The Monterey Security Architecture (MYSEA) Multilevel Security (MLS) Local Area Network (LAN) design specification document is an internal document containing sufficient detail to develop a prototype for the Trusted Path Server (TPS), Trusted Path Extension (TPE), Secure Session Daemon (SSD), Secure Session Server (SSS), Trusted Remote Session Server (TRSS), and supporting modules described in Chapters II and IV. The information in the design specification includes a list of constants (Section 3), a high-level description of the database and process modules (Sections 4 and 5), a description of the interfaces for each module (Section 6), and a detailed implementation specification for each module (Section 9). Appendix A of the design specification contains the C-language definition of the RAPSKT database.

The following is the table of contents from the detailed design specification document:

LIST OF REFERENCES

1. Defense Technical Information Center, "Joint Vision 2020", http://www.dtic.mil/jointvision/jvpub2.htm, March 2005.

2. Cynthia E. Irvine, Timothy E. Levin, Thuy D. Nguyen, David Shifflett, Jean Khosalim, Paul C. Clark, Albert Wong, Fancis Afinidad, David Bibighaus, Joseph Sears, "Overview of a High Assurance Architecture for Distributed Multilevel Security", *Proceedings of the 5th IEEE Systems, Man and Cybernetics Information Assurance Workshop*, United States Military Academy, West Point, NY, 10-11 June 2004, pg 38-45.

3. XTS-400, STOP 6.0, *Trusted Facility Manual*, Document ID: XTDOC0004-01, Getronics Government Solutions, LLC, Herndon, VA, August 2002.

4. XTS-400, STOP 6.0, *Programmer's Guide*, Document ID: XTDOC0006-01, Getronics Government Solutions, LLC, Herndon, VA, August 2002.

5. XTS-400, STOP 6.0, *User's Manual*, Document ID: XTDOC0005-01, Getronics Government Solutions, LLC, Herndon, VA, August 2002.

6. Susan BryerJoyner and Scott D. Heller, *Secure Local Area Network Services for a High Assurance Multilevel Network*, Master's thesis, Naval Postgraduate School, Monterey, CA, March 1999.

7. David Shifflett, *Multi-level Secure Local Area Network Project Design Document (Draft)*, Master's thesis, Naval Postgraduate School, Monterey, CA, October 2001.

8. Bradley R. Eads, *Developing a High Assurance Multilevel Mail Server*, Master's thesis, Naval Postgraduate School, March 1999.

9. Theresa Everette, *Examination of the Internet Message Access Protocol (IMAP) to Facilitate User-Friendly Multilevel Email Management*, Master's thesis, Naval Postgraduate School, Monterey, CA, September 2000.

10. Evelyn Louise Bersack, *Implementation of a Hypertext Transfer Protocol Server on a High Assurance Multilevel Secure Platform*, Master's thesis, Naval Postgraduate School, Monterey, CA, December 2000.

11. Emma J. M. Brown, *Facilitating Secure Mail in a High Assurance LAN*, Master's thesis, Naval Postgraduate School, Monterey, CA, September 2000.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, VA

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, CA

3. Hugo A. Badillo
   NSA
   Fort Meade, MD

4. George Bieber
   OSD
   Washington, DC

5. Deborah Cooper
   DC Associates, LLC
   Roslyn, VA

6. CDR Daniel L. Currie
   PMW 161
   San Diego, CA

7. CDR James Downey
   NAVSEA
   Washington, DC

8. Dr. Diana Gant
   National Science Foundation
   Arlington, VA

9. Richard Hale
   DISA
   Falls Church, VA

10. LCDR Scott D. Heller
    SPAWAR
    San Diego, CA

11. Wiley Jones
    OSD
    Washington, DC

12. Russell Jones
N641
Arlington, VA

13. David Ladd
Microsoft Corporation
Redmond, WA

14. Dr. Carl Landwehr
National Science Foundation
Arlington, VA

15. Steve LaFountain
NSA
Fort Meade, MD

16. Dr. Greg Larson
IDA
Alexandria, VA

17. Penny Lehtola
NSA
Fort Meade, MD

18. Ernest Lucier
Federal Aviation Administration
Washington, DC

19. CAPT Sheila McCoy
Headquarters U.S. Navy
Arlington, VA

20. Dr. Vic Maconachy
NSA
Fort Meade, MD

21. Doug Maughan
Department of Homeland Security
Washington, DC

22. Dr. John Monastra
Aerospace Corporation
Chantilly, VA

23. John Mildner
SPAWAR
Charleston, SC

24. Keith Schwalm
Good Harbor Consulting, LLC
Washington, DC

25. RADM Joseph Singer
NETWARCOM
Fort Meade, MD

26. Dr. Ralph Wachter
ONR
Arlington, VA

27. David Wirth
N641
Arlington, VA

28. Daniel Wolf
NSA
Fort Meade, MD

29. James Yerovi
National Reconnaisance Organization
Chantilly, VA

30. CAPT Robert Zellmann
CNO Staff N614
Arlington, VA

31. Dr. Cynthia E. Irvine
Naval Postgraduate School
Monterey, CA

32. Thuy Nguyen
Naval Postgraduate School
Monterey, CA

33. LT Robert C. Cooper
Naval Postgraduate School
Monterey, CA